REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 94	Technical

4. TITLE AND SUBTITLE
Load Balancing for the Parallel Adaptive Solution of Partial Differential Equations

5. FUNDING NUMBERS
DAAL03-91-G-0215

6. AUTHOR(S)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Rensselaer Polytechnic Institute
Troy, NY 12180-3590

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
ARO 29167.25-MA

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

ABSTRACT ON FIRST PAGE OF REPORT

14. SUBJECT TERMS

15. NUMBER OF PAGES
33

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Rensselaer

Department of Computer Science
Technical Report

Load Balancing for the Parallel Adaptive Solution of Partial Differential Equations

H. L. deCougny, K. D. Devine, J. E. Flaherty, R. M. Loy,
C. Ozturan, and M. S. Shephard

Scientific Computation Research Center
Rensselaer Polytechnic Institute
Troy, New York 12180-3590

Rensselaer Polytechnic Institute
Troy, New York 12180-3590

Report No. 94-8 **April 1994**

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF COLOR PAGES WHICH DO NOT REPRODUCE LEGIBLY ON BLACK AND WHITE MICROFICHE.

Load Balancing for the Parallel Adaptive Solution of Partial Differential Equations

H. L. deCougny, K. D. Devine, J. E. Flaherty, R. M. Loy,
C. Ozturan, and M. S. Shephard
Scientific Computation Research Center, Rensselaer Polytechnic Institute Troy, New York 12180-3590.

Dedicated to Bob Vichnevetsky on the occasion of his sixtieth birthday.

Abstract

An adaptive technique for a partial differential system automatically adjusts a computational mesh or varies the order of a numerical procedure with a goal of obtaining a solution satisfying prescribed accuracy criteria in an optimal fashion. Processor load imbalances will, therefore, be introduced at adaptive enrichment steps during the course of a parallel computation. We develop and describe three procedures for retaining and restoring load balance that have low unit cost and are appropriate for use in an adaptive solution environment.

Tiling balances loading by using local optimality criteria within overlapping processor neighborhoods. Elemental data are migrated between processors within the same neighborhoods to restore balance. Tiling can potentially be improved by creating a dynamic partition graph connecting processors and their neighboring regions. After coloring the edges of the graph, elemental data are transferred between processors by pairwise exchange.

Octree decomposition of a spatial domain is a successful three-dimensional mesh generation strategy. By performing tree traversals that (i) appraise subtree costs and (ii) partition spatial regions accordingly, we show that octree structures may also be used to balance processor loading.

Computational results are reported for two- and three-dimensional systems using nCUBE/2 hypercube, MasPar MP-2, and Thinking Machines CM-5 computers.

1 Introduction

Adaptive finite element methods that automatically refine or coarsen meshes (h-refinement) and/or vary the order of accuracy of a method (p-refinement) offer greater reliability, robustness, and computational efficiency than tradi-

tional numerical approaches for solving partial differential equations. High-order methods and the combination of mesh refinement and order variation (*hp*-refinement) can produce remarkably efficient methods with exponential convergence rates [2, 4, 5, 11, 12, 27]. Like adaptivity, parallel computation is making it possible to solve previously intractable problems. With problems continuing to increase in complexity through the inclusion of more realistic effects in models, it seems natural to unite adaptivity and parallelism to achieve the highest gains in efficiency. Adaptivity, however, introduces complications that do not arise when simpler solution strategies are implemented on parallel computers. Adaptive algorithms utilize unstructured [2] or hierarchical [3, 7] meshes that make the task of balancing processor loading much more difficult than with uniform structures. A balanced loading will, furthermore, become unbalanced as additional degrees of freedom are introduced or removed by adaptive *h*- and *p*-refinement.

Successful load partitioning strategies for unstructured-mesh computation on distributed-memory parallel computers employ recursive bisection to repeatedly split the discretized domain into two sub-domains having balanced loading. Specific techniques use geometric [6], connectivity [16], or spectral [26] information. When applied to the entire mesh, recursive bisection methods require a complete remapping of the elements of the mesh and, thus, involve a substantial overhead. Some methods also require considerable computation. Thus, global recursive bisection methods are too expensive for use with an adaptive method which, as noted, requires repeated mesh redistribution through the course of a computation. Recursive bisection may be of use with an adaptive strategy if applied locally to regions of the domain affected by adaptive enrichment [33].

Two partitioning strategies described herein use local migration to exchange elements between processors associated with neighboring spatial regions in order to achieve a global load balance. Local interchanges propagate incremental changes in the mesh or method between processors without solving an expensive global partitioning problem. Local computational cost metrics, such as the number of degrees of freedom, can be combined with similar information on partition boundaries to minimize the total workload including both the computational and communications efforts.

Our most mature partitioning strategy *tiling* [34] is a modification of a dynamic load balancing technique developed by Leiss and Reddy [25] that balances work within overlapping processor neighborhoods to achieve a global load balance. Work is migrated from a processor to others within the same neighborhood to obtain local optimality. We demonstrate the performance of tiling by using it with adaptive *h*- and *p*-refinement strategies to solve two-dimensional transient systems of conservation laws on a 256-processor nCUBE/2 hypercube (cf. Section 2).

Tiling can potentially be improved by creating a dynamic partition graph connecting processors and their neighboring regions. Loading information can be used to color edges of the partition graph so that work can be transferred between pairs of processors in a manner similar to a pairwise heuristic strategy introduced by Hammond [19]. This strategy, described in Section 3, distributes load imbalances more quickly than tiling and simplifies the task of minimizing a partition's communications volume. Two-dimensional computations performed on a MasPar MP-2 SIMD system demonstrate some capabilities of this procedure.

Octree decomposition is a successful strategy for generating three-dimensional unstructured meshes [29] and we develop (cf. Section 4) a partitioning technique that exploits the properties of tree-structured meshes. Partitioning may be done locally or globally, but, in either case, it is inexpensive and, hence, may be with adaptive procedures. Partitioning is based on two tree traversals that (i) calculate the processing costs of subtrees connected to each node and (ii) form the partitions. When used globally, partitions have approximately the same communications volume as other strategies [21, 24, 26], but their cost is far less. We demonstrate the performance of the tree-based partitioning technique on three-dimensional meshes that are associated with flight vehicle flows. Results computed on a Thinking Machines CM-5 computer are presented for an adaptive h-refinement solution of the Euler equations for a supersonic conical flow.

2 Tiling

2.1 Adaptive Enrichment

We describe adaptive h- and p-refinement local time-stepping algorithms that are being used with the tiling partitioning scheme (Section 2.2) but which are typical of adaptive strategies. Applied to vector systems of conservation laws, finite element solutions $\mathbf{U}(\mathbf{x}, t)$ are obtained on a two-dimensional net of rectangular elements using a spatially discontinuous Galerkin method [5, 8, 9, 10] and explicit Runge-Kutta integration [5]. A spatial discretization error estimate $E(t)$ in the L^1 norm is obtained by p-refinement [5, 11] and used to control adaptive spatial enrichment so that $E(t) \leq \epsilon$, for a prescribed tolerance ϵ.

With adaptive p-refinement (cf. Figure 1) , we initialize $\mathbf{U}(\mathbf{x}, 0)$ to the lowest-degree polynomial satisfying $E_j(0) \leq \epsilon/J$, $j = 1, 2, \ldots, J$, where $E_j(t)$ is the restriction of $E(t)$ to element j and J is the number of elements in the mesh. After each time step, we compute E_j, $j = 1, 2, \ldots, J$, and increase the polynomial degree of $\mathbf{U}$ on element j by one if $E_j > \epsilon/J$ $(= TOL)$. The solution $\mathbf{U}$ and the error estimate are recomputed on enriched elements, and further increases of degree occur until $E_j \leq TOL$ on all elements. The need for back-

```
void adaptive_p_refinement()
{
    while (t < t_final) {
        perform_runge_kutta_time_step(all_elements);
        do {
            Solution_Accepted = TRUE;
            for each element {
                error_estimate = calculate_estimate();
                if (error_estimate > TOL) {
                    mark_element_as_unnacceptable();
                    increase_elements_polynomial_degree();
                    Solution_Accepted = FALSE;
                }
            }
            if (!Solution_Accepted) {
                recalculate_solution_on_unacceptable_elements();
            }
        } while (!Solution_Accepted);
        accept_solution(all_elements);

        predict_degrees_for_next_time_step(all_elements);
        t = t + Δt;
    }
}
```

Figure 1: An adaptive p-refinement procedure.

tracking may be reduced by predicting the degree of the approximation needed to satisfy the accuracy requirements for the subsequent time step. After a time step is accepted, if $E_j > H_{max}\,TOL$, $H_{max} \in (0,1]$, we increase the degree of $\mathbf{U}(t+\Delta t)$ on element j for the next time step. If $E_j < H_{min}\,TOL$, $H_{min} \in [0,1)$, we decrease the degree of $\mathbf{U}(t+\Delta t)$ for the next time step.

In the adaptive h-refinement method, we locally refine element j if $E_j > TOL/2^m$, where m is the level of refinement. Refinement involves dividing an element into four and initializing the solution through L^2-projection of the coarse data [5]. Elements neighboring high-error elements are also refined to provide a buffer-zone between high- and low-error regions and maintain a difference of at most one level of refinement across element edges. For each time step, the local finite element method [5] is applied on successively finer meshes. To satisfy the Courant conditions, the time step is halved on each finer mesh. An outline of the h-refinement algorithm is shown in Figure 2.

```
void adaptive_h_refinement(mesh, t_start, t_final, Δt)
{
    t = t_start;
    while (t < t_final) {
        perform_runge_kutta_time_step(all elements of mesh);
        for each element of mesh {
            error_estimate = calculate_estimate();
            fine_mesh = mesh → nextmesh;
            if ((error_estimate > TOL) && (element_not_refined_yet)) {
                refine_element_into_four_fine_elements();
                add_new_elements(fine_mesh);
            }
        }
        if (mesh is refined) {
            buffer(fine_mesh);
            project_coarse_data(mesh, fine_mesh);
            adaptive_h_refinement(fine_mesh, t, t + Δt, Δt/2);
            interpolate_fine_solution_to_coarse_mesh(fine_mesh, mesh);
        }
        t = t + Δt;
    }
}
```

Figure 2: An adaptive h-refinement procedure.

2.2 Dynamic Load Balancing via Tiling

As noted, tiling is a modification of a load balancing technique of Leiss and Reddy [25, 28] who used local optimality criteria within overlapping neighborhoods. A neighborhood consisted of a processor at the center of a cricle of a given radius and all processors within that circle. With tiling, we extend the definition of a neighborhood to include all processors having finite elements that are neighbors of elements in the central processor (cf. Figure 3). Every processor is the center of one neighborhood, and may belong to many neighborhoods. Elements are migrated only to processors having neighbors of the migrating elements.

The tiling algorithm consists of (*i*) a computation phase and (*ii*) a balancing phase, and is designed to be independent of the application. The computation phase corresponds to the application's implementation without load balancing. Each processor operates on its local data, exchanges inter-processor boundary data, and processes the boundary data. A balancing phase restores load balance following a given number of computation phases. Each balancing phase

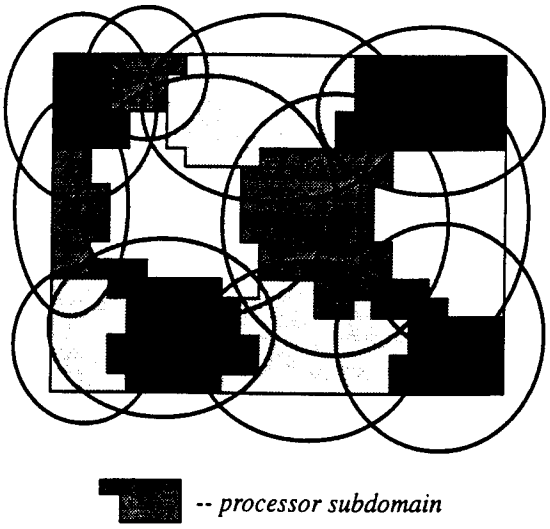


Figure 3: Example of 12 processors in 12 neighborhoods using tiling.

consists of the following operations:

1. Each processor determines its work load as the time to process its local data since the previous balancing phase less the time to exchange inter-processor boundary data during the computation phase. Neighborhood average work loads are also calculated.

2. Each processor compares its work load to the work load of the other processors in its neighborhood and determines those processors having loads greater than its own. If any are found, it selects the one with the greatest work load (ties are broken arbitrarily) and sends a request for work to that processor. Each processor may send only one work request, but a single processor may receive several work requests.

3. Each processor prioritizes the work requests it receives based on the request size, and determines which elements to export to the requesting processor. Details of the selection algorithm are given below.

4. Once elements to be exported have been selected, the importing processors and processors containing neighbors of the exported elements are notified. Importing processors allocate space for the incoming elements, and the elements are transferred.

Each processor knows the number of computation phases to perform before entering the balancing phase. Synchronization guarantees that all processors enter the balancing phase at the same time.

The technique for selecting elements gives priority to elements with neighbors in the importing processor to prevent the creation of "narrow, deep holes" in the element structures. Elements are assigned priorities (initially zero) based upon the locality of their neighbors. An element's priority is decreased by one for each neighbor in its own processor, increased by two for each neighbor in the importing processor, and decreased by two for each neighbor in some other processor. Thus, elements whose neighbors are already in the importing processor are more likely to be exported to that processor than elements whose neighbors are in the exporting processor or some other processor. When an element has no neighboring elements in its local processor, it is advantageous to export it to any processor having its neighbors. Thus, "orphaned" elements are given the highest export priority.

Because individual elements' processing costs can vary widely in the adaptive p-refinement method, elemental processing costs are computed and used so that the minimum number of elements satisfying the work request are exported. This approach differs from Wheat [34], where the average cost per element is used to determine the number of export elements. When two or more elements have the same priority, the processor selects the element with the largest work load that does not cause the exported work to exceed the work request or the work available for export.

In the adaptive h-refinement method, the local time-stepping scheme outlined in Figure 2 requires that each mesh level be distributed evenly over the processor array to avoid idle time. Communication costs are increased since offspring elements may be on different processors than their parent elements; however, the increase in communication time is outweighed by a decrease in processor idle time. Memory overhead for the tiling algorithm is also increased, as processor location information for parent and offspring elements must be maintained and ghost elements must be allocated for non-local coarse elements along coarse-fine mesh interfaces. Selection priority schemes which account for the interconnections between mesh levels could reduce the additional communication and storage needed; such schemes are the subject of future study.

Example 2.1 We solve

$$u_t + 2u_x + 2u_y = 0, \quad t > 0, \tag{1a}$$

on $0 < x, y < 1$ with initial and boundary conditions specified so that

$$u(x, y, t) = \frac{1}{2}[1 - \tanh(20x - 10y - 20t + 5)], \tag{1b}$$

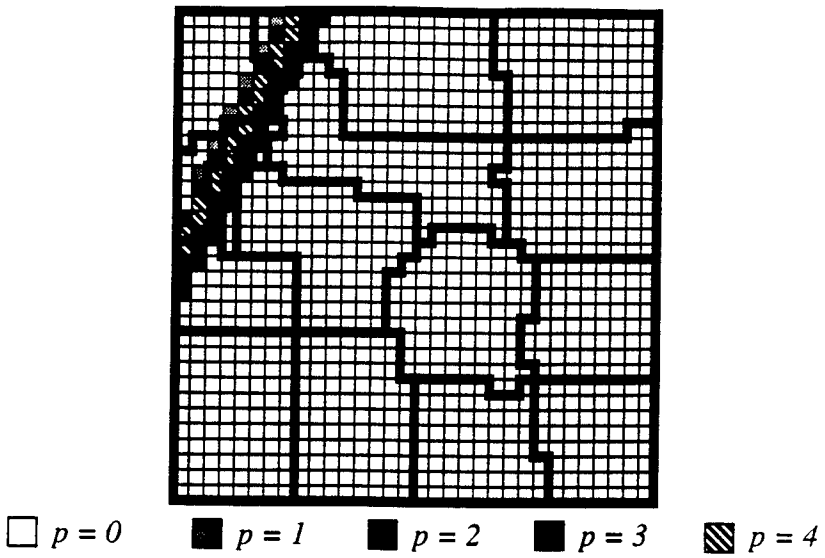


Figure 4: Processor domain decomposition after 20 time steps for Example 2.1 using adaptive p-refinement and tiling. Dark lines represent processor subdomain boundaries.

using adaptive p-refinement on a 32×32-element mesh with $TOL = 3.5 \times 10^{-5}$ and tiling on 16 processors. In Figure 4 we show the processor domain decomposition after 20 time steps. The shaded elements have higher-degree approximations and, thus, higher work loads. The tiling algorithm redistributes the work so that processors with high-order approximations have fewer elements than those processors with low-order approximations. The total processing time for the adaptive p-refinement method was reduced 41.98% from 63.94 seconds to 37.10 seconds by balancing once each time step. The average/maximum processor work ratio without balancing is 0.362, and with balancing, it is 0.695. Parallel efficiency is increased from 35.10% without balancing to 60.51% with tiling.

We also solve (1) using the adaptive h-refinement method on a 16×16-element base mesh with $TOL = 1.0 \times 10^{-3}$ and tiling on 4 processors. In Figure 5 we show the processor domain decomposition after 10 times steps. The decomposition is shown for each mesh level. The total processing time for the adaptive h-refinement method was reduced 58.0% from 104.89 seconds to 44.01 seconds by balancing a mesh after each time step on the mesh. The average/maximum processor work ratio without balancing is 0.271, and with balancing, it is 0.747. Parallel efficiency is increased from 26.7% without balancing to 63.8% with tiling.

Example 2.2 Again, we solve (1) on $\Omega = (0, 16) \times (-7.5, 8.5)$ with a fixed-order method ($p = 2$) and the adaptive p-refinement method for 86 time steps to $t = 0.3$ using a 160×160-element mesh on 256 processors of the nCUBE/2

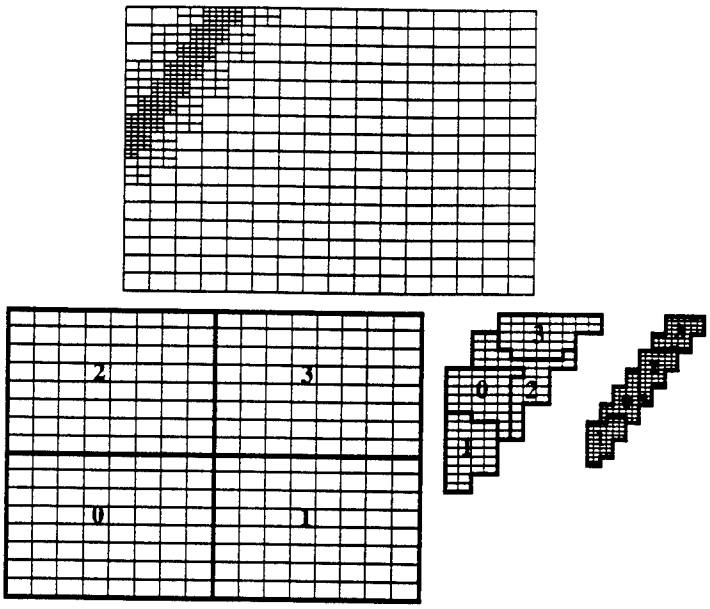


Figure 5: Processor domain decomposition after 10 time steps for Example 2.1 using adaptive h-refinement and tiling. The decomposition on each mesh level is shown.

without balancing and with balancing once each time step. In the adaptive p-refinement method, polynomial degrees of the elements varied from 0 to 2, and computation time per element varied from 0.02 to 1.2 seconds per time step, indicating a great deal of imbalance along the front. Even without balancing, the adaptive p-refinement method required 65.7% less execution time than the fixed-order method to achieve comparable accuracy (cf. Table 1). With balancing, the maximum computation time of the adaptive method (not including communication or balancing time) was further reduced by 78.1%. The irregular subdomain boundaries created by the tiling algorithm increased the average communication time by 40.7%. Despite the extra communication time and the load balancing time, however, we see a 70.1% improvement in the total execution time over the non-balanced adaptive method, and 89.7% over the fixed-order method.

Example 2.3 We solve

$$u_t + 2u_x + 2u_y = 0, \quad t > 0, \tag{2a}$$

	Adaptive p-refinement		Fixed-order
	Without Tiling	With Tiling	Without Tiling
Total Execution Time (seconds)	3636.53	1088.36	10,590.87
Max. Computation Time (seconds)	3557.77	778.57	10,570.07
Average/Maximum Work Ratio	0.118	0.543	0.999
Avg. Communication Time (seconds)	22.31	31.38	19.03
Max. Balancing Time (seconds)	0.00	28.98	0.00
Parallel Efficiency	11.62%	38.84%	99.71%

Table 1: Performance comparison for Example 2.2 using adaptive p-refinement without balancing and with balancing at each time step, and a fixed-order method yielding comparable accuracy.

on $\Omega = (0, 16) \times (-7.5, 8.5)$ with initial and boundary conditions specified so that

$$u(x, y, t) = \frac{1}{2}(1 - \tanh(100x - 10y - 20t + 5), \tag{2b}$$

with $p = 0$ on a uniform 640×640-element mesh and on a 160×160-element base mesh with adaptive h-refinement for 60 time steps on 256 processors without balancing and with balancing once after each time step on each mesh level. Two levels of refinement were used along the steep front. We compare the adaptive h-refinement computation with a uniform-mesh computation of similar accuracy. The adaptive solution required 46.9% less total execution time than the non-adaptive solution, despite the load imbalances created by the adaptive method (cf Table 2). We further reduce the execution time by combining balancing with the adaptive h-refinement method. With balancing, the maximum computation time (not including communication or balancing time) was reduced by 86.1% over the adaptive method without balancing. The average communication time with balancing is nearly doubled, due to the communication between coarse and fine mesh elements that have been migrated to different processors. The total balancing time is larger than the balancing time for p-refinement, since many more balancing phases are used as each mesh level is load balanced. Despite the tiling overhead, however, we see an 80.3% improvement in the total execution time of the h-refinement method.

	Adaptive h-refinement		Uniform Mesh
	Without Tiling	With Tiling	Without Tiling
Total Execution Time (seconds)	3455.07	681.48	6508.16
Max. Computation Time (seconds)	3430.17	476.00	6491.07
Average/Maximum Work Ratio	0.0743	0.535	1.000
Avg. Communication Time (seconds)	7.68	15.33	14.82
Max. Balancing Time (seconds)	0.00	42.14	0.00
Parallel Efficiency	7.38%	37.36%	99.70%

Table 2: Performance comparison for Example 2.3 using adaptive h-refinement without balancing and with balancing at each time step on each mesh level, and a uniform mesh yielding comparable accuracy.

3 Element Redistribution by Pairwise Exchange

3.1 Redistribution

The element redistribution algorithm and its similarities and differences to the tiling procedure of Section 2 are described through an example. Consider the unbalanced mesh distribution over eleven processors as shown in Figure 6(a). Let $G_P(V, E)$ be a *partition graph* with each vertex in V representing a partition assigned to a processor and E representing the set of edges between partitions. Two partitions u and v are connected by an edge $(u, v) \in E$ if they share a mesh edge. If two partitions share only a mesh vertex, then they are not considered adjacent in the partition graph. The reason for the mesh edge connectivity requirement between partitions in G_P is twofold. First, by excluding vertex adjacency, the number of edges in E and, hence, the time to communicate with adjacent processors is kept minimal. Second, transferring elements between partitions that share only a vertex results in a higher surface to volume ratio which increases communication cost. Figure 6(b) shows the partition graph obtained from the mesh distribution in Figure 6(a).

Following Leiss and Reddy [25], a workload deficient processor will request work from its most heavily loaded neighbor. As a result, a processor can receive multiple requests but can only request load from one processor. This

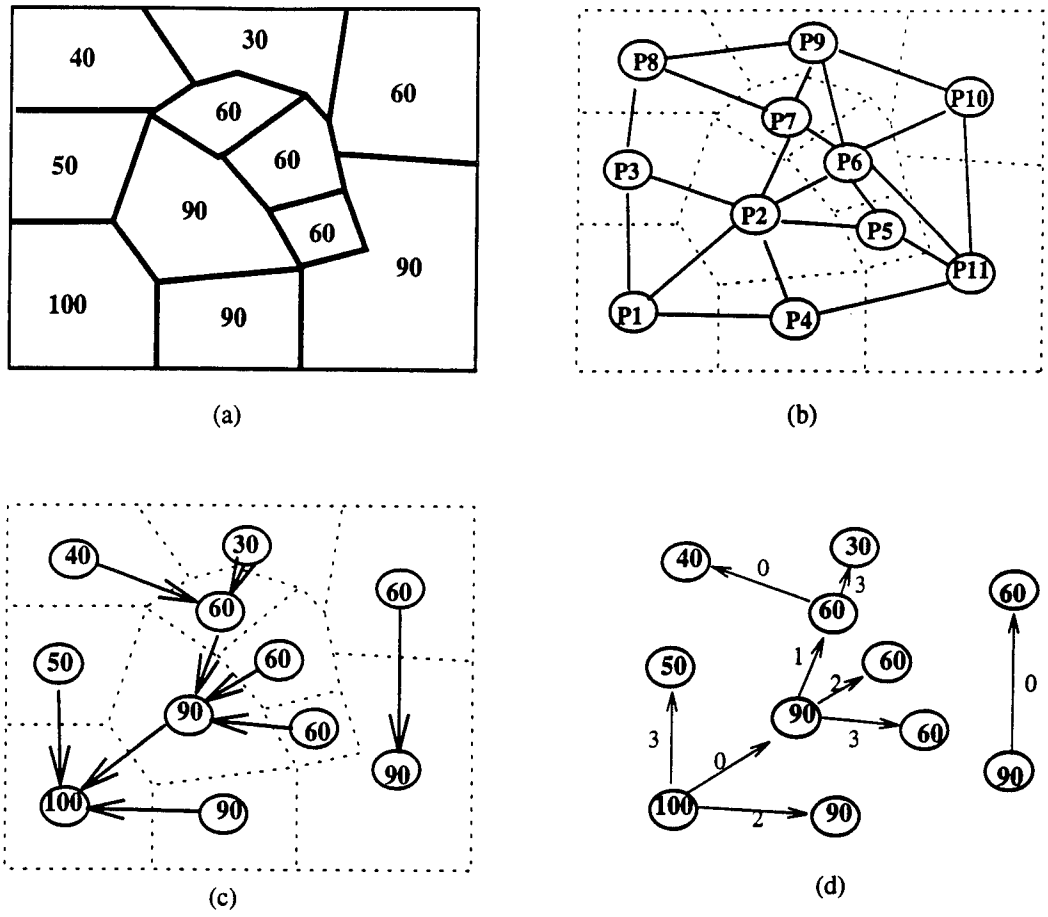


Figure 6: (a) Unbalanced load in each partition, (b) partition graph P_G, (c) load request, (d) load transfer between pairs in steps 0,1,2,3.

pattern of requests produces a load hierarchy and forms a forest of trees T_i as shown in Figure 6(c). The trees T_i in the forest are subgraphs of the partition graph G_P.

The current proposed algorithm for redistribution pairs the processors on each tree T_i and transfers load from the heavily loaded pair to the other. The pairing of processors is equivalent to coloring the edges of each tree T_i with colors representing separate load transfer (communication) cycles. The edge coloring approach synchronizes the load transfer between neighboring processors and differs from the approach of Section 2. In this implementation, processors P_2 and P_7 in Figure 6(b) would be engaged with load transfer with only one neighbor at a single transfer step. With tiling, however, processors P_2 and P_7 would receive and send work during the same transfer step. Tiling would be more efficient if there were load transfer in one direction only, i.e.,

from heavily loaded to less loaded processor. Processors P_2 and P_7 would be doing useful work packing elements to be transferred to their offspring while their parents pack elements to be transferred to them. With the suggested exchange, processors P_2 and P_7 would remain idle waiting for load transfer from parents and would transfer load to offspring after the transfer from the parent has been completed. However, this disadvantage is overshadowed by a number of advantages. First, a smaller number of messages and the synchronous transfer of load increase communication performance. Second, since the processors are synchronized by pairs, a greater repertoire of selection criteria can be used to decide which elements to transfer. Unlike tiling, where elements can only be transferred from heavily loaded to less loaded processors, the pairing allows elements to be transferred from the less loaded to the heavily loaded pair. This can be useful in improving the surface to volume ratio of the partitions. Since there is no explicit synchronization by edge coloring with tiling, a bidirectional transfer of load would be extremely difficult.

Figure 6(d) shows the coloring phase used to pair the processors. If $\Delta(G)$ denotes the maximum vertex degree (number of edges incident on a vertex) in a graph G, then Vizing's theorem [36] indicates that the graph G can be edge-colored using C colors where $\Delta(G) \leq C \leq \Delta(G) + 1$. For some special graphs including the trees the number of colors needed is exactly $\Delta(G)$. Therefore, $\Delta(T_i)$ colors are required to color the tree T_i.

The main steps of the *redistribute* algorithm are illustrated in Figure 7 and the detailed steps follow:

1. The transfer of work between paired processors is iterated until the load on each processor converges to a value close to the optimal balance (cf. Section 3.2).

2. Load differences are computed by having each processor send a load value to its neighbors and correspondingly receive load values from its neighbors. This step takes $\Delta(G_P)$ time.

3. The Leiss and Reddy [25] load request process is invoked and this results in the forest of trees T_i. An edge of G_p is simply marked when a request has been made indicating whether or not it is a tree edge. Since the incoming requests for load should be sorted, this step takes $O(\max_i\{d_i \cdot \log d_i\})$ time where $d_i = \Delta(T_i)$.

4. Deciding how much load to transfer to requesting processors is crucial in making the redistribution algorithm to converge. Criteria for this step are given with convergence criteria in Section 3.2.

5. To facilitate efficient parallel scan operations on the trees T_i, each tree is linearized by setting links between neighboring processors. The links

```
void redistribute(mesh,tol_imbalance,max_iters)
{
    mypid = get_my_processor_id();
    iter = 0 ;
    while ( imbalance(mesh) > tol_imbalance && iter < max_iters) {
        compute_neighboring_load_differences(mesh);
        proc = neighbor_having_largest_load_difference(mesh);
        T = request_load_from_neighbor(proc,mesh);
        determine_amount_of_load_to_send_or_receive(T,mesh);
        set_up_links_to_linearize_tree(T);
        color_tree(T);

        for each color C {
            if (processor_owns_ color(C,myid) &&
                is_a_neighbor_of_color_pair(C,mypid,pair_processor))
              transfer_load_between_pair(mesh,mypid,pair_processor);
            }
        iter = iter + 1;
        }
}
```

Figure 7: Redistribution algorithm.

can be constructed by either defining an *Euler Tour* [22] or a depth first traversal [31] of the tree.

6. The linearized tree is edge-colored by employing a parallel scan operation. Since there can be up to $\Delta(T_i)$ links on a processor, the scan operation using the Euler Tour links takes $O(\max_i\{d_i \cdot \log|V_i|\})$ where $|V_i|$ denotes the number of vertices in tree T_i. The use of depth first traversal links stores 2 links per processor and hence enables a more efficient scanning step with complexity $O(\max_i\{\log|V_i|\})$.

7. The steps of the load transfer are synchronized by the edge coloring of the tree. One iteration of the redistribution algorithm involves C steps corresponding to the $\max_i(\Delta(T_i))$ colors. Note that this synchronization also allows for bi-directional transfer of load between pair processors.

8. The elements to be transferred are selected. As with tiling, a cost is associated with the partition boundary elements. This cost reflects the communication as well as the computational cost of the element. The

element that will yield the smallest increase in communication cost will be transferred.

3.2 Load Transfer and Convergence

Suppose a parent processor with load value L_0 has m load requesting offspring with load values L_i, $i = 1, 2, \ldots, m$, as shown in Figure 8(a). Each offspring requests an amount r_i which is equal to the difference from its current load to the average of its and its parent's load, i.e.,

$$r_i = \lceil (L_0 - L_i)/2 \rceil. \tag{3}$$

The parent processor will decide to send a total amount which will make its load become the average of the loads L_i, $i = 0, 1, \ldots, m$,

$$to\_send_{tot} = L_0 - \frac{\sum_{i=0}^{m} L_i}{m+1}.$$

The parent determines the individual amounts to_send_i to transfer to children in proportion to the their load request:

$$to\_send_i = min\{r_i, to\_send_{tot} \cdot \frac{r_i}{\sum_{j=1}^{m} r_j}\}.$$

The minimum of the two values is taken in order to prevent transferring loads greater than the requested load. Figure 8(b) shows the load requests and load grants for a subtree in the redistribution example.

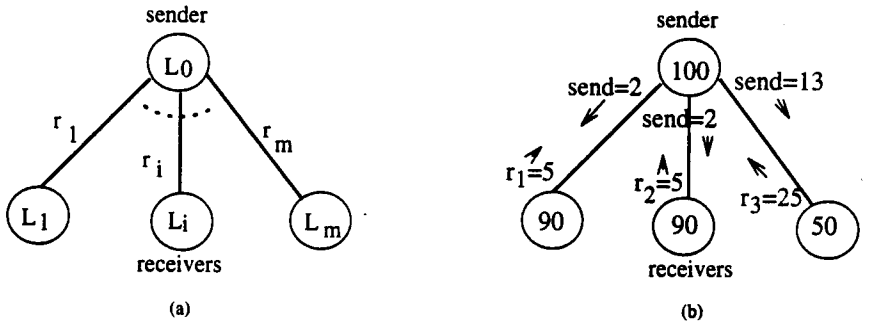


Figure 8: (a) Load request $r_i = \lceil (L_0 - L_i)/2 \rceil$ from sender and (b) an example of transferred amounts.

Convergence of the iterative load balancing algorithm without limit cycles (indefinite repeated load transfer patterns) was investigated by Leiss and Reddy [25]. Let an *H-neighborhood* denote the neighbors of a processor within

a distance of H and C denote a load threshold value. If elements are taken as load units, then $C = 1$. Also let d indicate the *diameter* (the maximum of all the shortest paths between any two nodes) of the processor graph which in the present case is the partition graph. Finally, define *H-neighborhood* imbalance at time t as the variance

$$GIMB_H^t = \sum_{p \in P} (L(p) - \alpha)^2.$$

Here P is the processor set, $L(p)$ is the load value on processor p, and α is the average load value per processor. Leiss and Reddy show that

1. after a rebalancing iteration $GIMB_H^{t+1} < GIMB_H^t$ and
2. after balancing terminates, the maximum imbalance in the whole system is bounded by $\lceil \frac{d}{2H} \cdot C \rceil$.

According to the first result, the imbalance in the *neighborhood* and *not* necessarily the whole system will reach a minimum since $GIMB_H^t$ is a decreasing function of t. The second result states that if the system is neighborhood-balanced, the whole system can still be severely imbalanced. A worst case example is the configuration with n processors forming a one-dimensional chain and each having a load that differs from neighbor only by $L_{i+1} - L_i = 1$, i.e., a load ramp. If $H = 1$ and $C = 1$, then, since $d = n$, the imbalance after termination of the algorithm will be $n/2$. Increasing the neighborhood measure H to $n/2$ will balance the system globally. However, $H = n/2$ will require *each* of the n processors to send messages to the $n/2$ H-neighbors. Hence, choosing $H = n/2$ is impractical. In general, the case $H > 1$ will increase the communication volume and hence make the iterative balancing algorithm inefficient.

To avoid this problem with Leiss and Reddy's [25] approach while keeping $H = 1$, two modifications are made to handle the case when the load difference between the neighboring processors is C. Unlike tiling, which sends at most $\lfloor (L_0 - L_i)/2 \rfloor$ work units and considers $L_0 - L_i = 1$ as balanced, the current procedure exchanges the excess load as given by Eq (3) even if $GIMB$ remains the same. Hence, it allows the case when $GIMB_H^{t+1} \leq GIMB_H^t$. The previous exchange is stored to ensure that excess load is not transferred back to the original processor preventing period two limit cycles. This period modification does not, however, avoid cycles of period greater than two.

Example 3.1 The iterative redistribution heuristic was tested on the MasPar MP-2 system which has a torus connected architecture and a SIMD style of computation. Up to 2048 processors were used in the test cases with each processor having 64K bytes of memory. The MasPar system provides two types

of communication mechanisms. The *xnet* mechanism provides a fast eight-way communication between processors arranged in a mesh topology. The *router* provides a slower general purpose communication among any pair of processors. Since our applications involve highly unstructured meshes with curved boundaries, the communication requirements between mapped partitions are irregular. Hence, the slower router communication had to be chosen for the implementation.

Four test cases involving meshes on a square and an irregular region were run. Starting with a coarse mesh, *Orthogonal Recursive Bisection* [6] was used to get an initial partition. The partitioned mesh was mapped onto the processors and refined selectively in parallel to create imbalanced processor loads. The square mesh was refined in one corner to create a "plateau" of high load distribution. As the neighboring load transfers progress, the plateau evolves to the difficult redistribution example involving a ramp load distribution. Table 3 shows various statistics for the test cases. In square1, a small mesh with 16 processors was employed (cf. Figure 9). In square2 and square3, 2048 processors were employed with refinement in 4 and 16 processors respectively in the upper right corner of the mesh as shown in Figure 10(a). The final test involved a highly unstructured mesh with a curved boundary (Figure 11).

Test	Number of elements	Number of processors	Average elements per processor	Load (Min,Max) before	Load (Min,Max) after	Max boundary edges before	Max boundary edges after	Number of iterations	Time (secs)
square1	164	16	10.25	2, 32	7,11	16	12	25	9.7
square2	32904	2048	16.06	16,52	16,18	24	21	63	33.6
square3	33300	2048	16.25	16,52	16,18	24	25	398	214.9
curved	1008	32	31.5	18,47	31,32	22	25	25	12.3

Table 3: Test cases and various statistics before and after convergence to load balance

In Table 3, we list the average number of elements per processor, the maximum and minimum loads and the maximum number of edges located on the boundary of the partitions before and after the redistribution algorithm has been run until convergence to optimal load balance. Tests square1, square2, and curved show good convergence results. Test square3, on the other hand, shows slow convergence even though the number of elements and the maximum imbalance is similar to the square2 test. Since a ramp evolves during redistribution, the amount of load that can be transferred from the high load plateau to the less loaded processors is small. In the worst case of a one-dimensional ramp with a unit load difference between neighboring processors, the maximum number of elements that can be transferred per iteration is one. Hence, the larger the excess load to be migrated from the plateau, the slower

the convergence of the redistribution algorithm. The average load and the distance the elements have to travel is approximately the same in both the square2 and square3 test cases. Since square3 has four times the excess load of square2, we would expect the number of iterations of square3 to be around four times that of square2. The convergence history, shown in Figures 10(b) and 10(c), point to this effect.

The number of boundary edges, which represents the communications volume of a partition, does not necessarily decrease after redistribution. Whereas tests square1 and square2 showed a slight reduction, square2 and curved meshes showed an increase in the number of boundary edges after balancing. Hence, even though the redistribution algorithm performs well in reducing the imbalance, the selection criteria used for element migration does not guarantee reduction of communication costs.

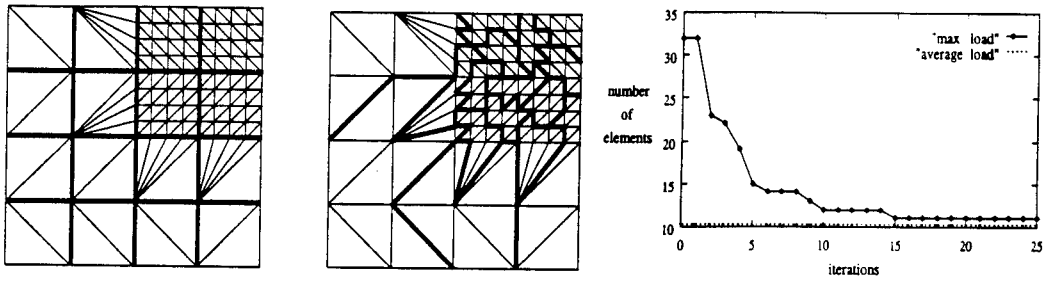


Figure 9: Test square1 of Example 3.1: (a) unbalanced load after mesh refinement, (b) after redistribution, and (c) convergence history.

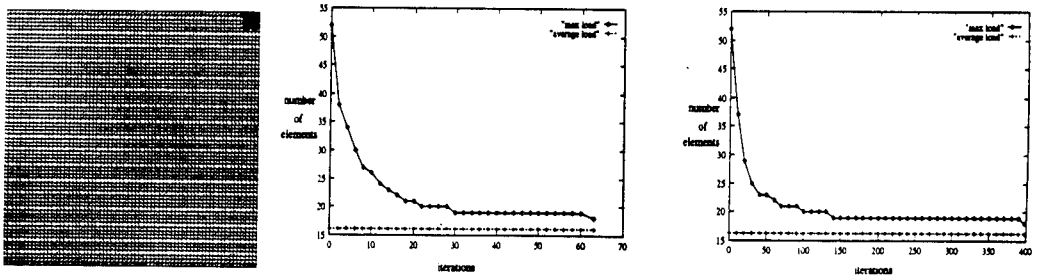
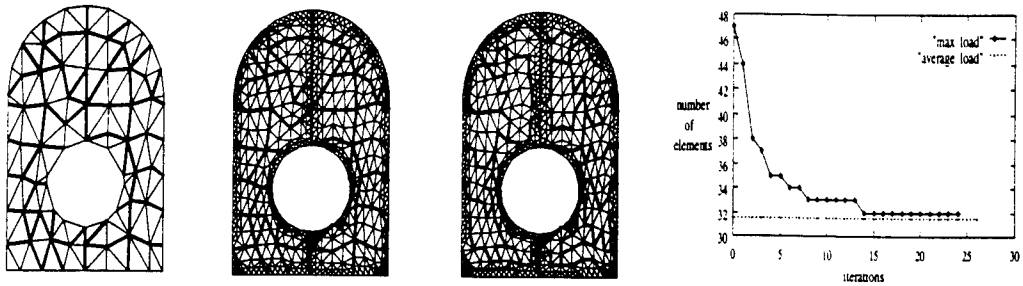


Figure 10: (a) Test mesh for square2 and square3 of Example 3.1, (b) and (c) corresponding convergence histories.

The convergence history plots given in Figures 9-11 show a sharp drop in the imbalance within the first few iterations and slower drops in later iterations. We further demonstrate the performance of the heuristic in Table 4 by listing the number of iterations and the cpu time required to achieve 50%,75% and



(a) (b) (c) (d)

Figure 11: Test curved of Example 3.1. (a) initial ORB partitioned coarse mesh, (b) unbalanced mesh after refinement, (c) balanced mesh after redistribution, and (d) convergence history.

90% reduction in the original imbalance. Since higher percentage reductions require far more iterations, a trade-off can be established between the time to do a computation with an imbalanced load and the time needed to achieve further reductions in imbalance. As a result, the redistribution process can be stopped at a smaller number of iterations.

	Percent Reduction in Imbalance					
	Iterations			Time(secs)		
Test	50 %	75 %	90 %	50 %	75 %	90 %
square1	4	5	10	5.2	5.7	7.6
square2	4	11	30	7.1	12.7	21.9
square3	13	29	98	28.6	47.7	92.8
curved	2	4	9	4.4	6.4	8.9

Table 4: Iterations and cpu times to achieve various percent reductions in imbalance for Example 3.1.

4 Octree-Based Partitioning

We describe a tree-based partitioning technique that utilizes the hierarchical structure of octree-derived unstructured meshes to distribute elemental data across processors' memories while reducing the amount of data that must be exchanged between processors. An octree-based mesh generator [29] recursively subdivides an embedding of the problem domain in a cubic universe into eight octants wherever more resolution is required. Octant bisection is initially based on geometric features of the domain but solution-based criteria are introduced during adaptive h-refinement. Finite element meshes of

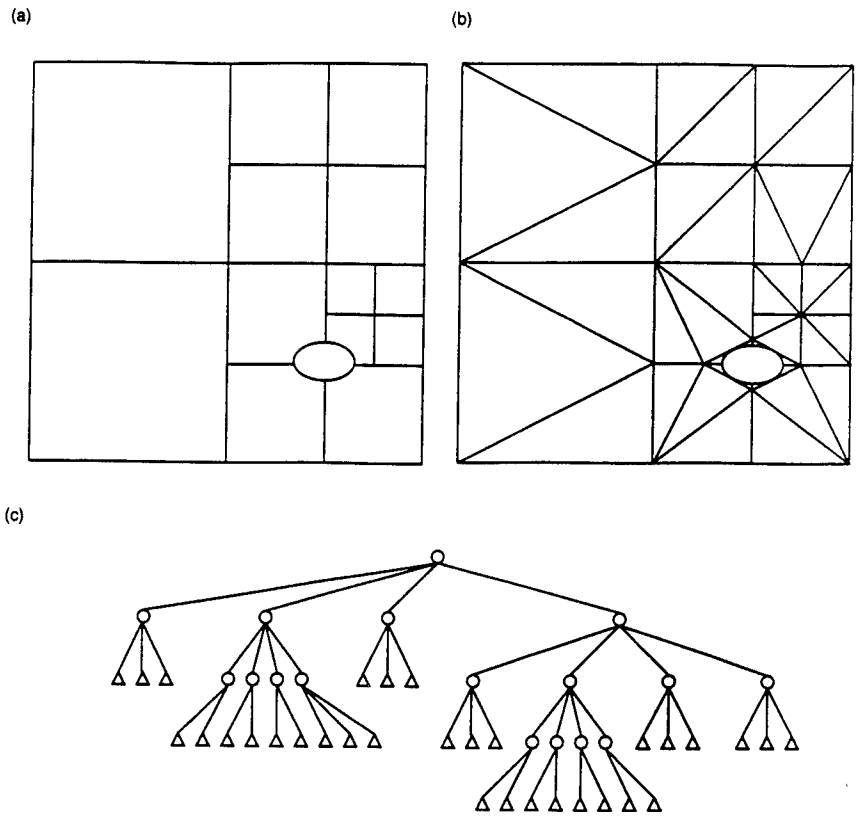


Figure 12: (a) A quadtree representation of the flow field surrounding an object, (b) division of terminal quadrants into triangular elements, and (c) quadtree structure.

tetrahedral elements are generated from the octree by subdividing terminal octants.

In Figure 12, we illustrate the tree and mesh for a two-dimensional flow domain containing a small object. The root of the tree represents the entire domain (Figure 12(c)). The domain is recursively quartered until an adequate resolution of the object is obtained (Figure 12(a)). A smooth gradation is maintained by enforcing a one-level maximum difference between adjacent quadrants. After appropriate resolution is obtained, leaf quadrants are subdivided into triangular elements that are pointed to by leaf nodes of the tree (Figures 12(b,c)). Quadrants containing the object are decomposed using the geometry of the object. Smoothing [29], which normally follows element creation, is not shown.

Our tree-based based partitioning algorithm creates a one-dimensional ordering of the octree and divides it into nearly equal-sized segments based on tree topology. The first step of the algorithm is the determination of cost metrics of all subtrees. Cost is currently defined as the number of elements

within a subtree. For a leaf octant, this would simply be the number of tetrahedra associated with it. P-refinement would necessitate the inclusion of an element's order into the cost function. If the solution algorithm employs spatially-dependent time steps then, typically, a greater number of smaller time steps must be taken on smaller elements and this must also be reflected in the subtree cost. In any event, appropriate costs may be determined by a postorder traversal of the octree.

The second phase of the partitioning algorithm uses the cost information to construct the actual partitions. Since the number of partitions is prescribed and the total cost is known from the first phase, we also know the optimal size of each partition. Partitions consist of a set of octants that are each the root of a subtree and are determined by a truncated depth-first search. Thus, octree nodes are visited in depth-first order, and subtrees are accumulated into successive partitions. The subtree rooted at the visited node is added to the current partition if it fits. If it would exceed the optimal size of the current partition, a decision must be made as to whether it should be added, or whether the traversal should examine it further. In the latter case, the traversal continues with the offspring of the node and the subtree may be divided among two or more partitions. The decision on whether to add the subtree or examine it further is based on the amount by which the optimal partition size is exceeded. A small excess may not justify an extensive search and may be used to balance some other partition which is slightly undersized. When the excess at a node is too large to justify inclusion in the current partition, and the node is either terminal or sufficiently deep in the tree, the partition is closed and subsequent nodes are added to the next partition.

This partitioning method requires storage for nonterminal nodes of the tree which would normally not be necessary since they contain no solution data. However, only minimal storage costs are incurred since information is only required for tree connectivity and the cost metric. For this modest investment, we have a partitioning algorithm that only requires $O(J)$ serial steps.

Partitions formed by this procedure do not necessarily form a single connected component; however, the octree decomposition and the orderly tree traversal tend to group neighboring subtrees together. Furthermore, a single connected component is added to the partition whenever a subtree fits within the partition.

A tree-partitioning example is illustrated in Figure 13. All subtree costs are determined by a post order traversal of the tree. The partition creation traversal starts at the root, Node 0 (Figure 13(a)). The node currently under investigation is identified by a double circle. The cost of the root exceeds the optimal partition cost, so the traversal descends to Node 1 (Figure 13(b)). As shown, the cost of the subtree rooted at node 1 is smaller than the optimal partition size and, hence, this subtree is added to the current partition, p0,

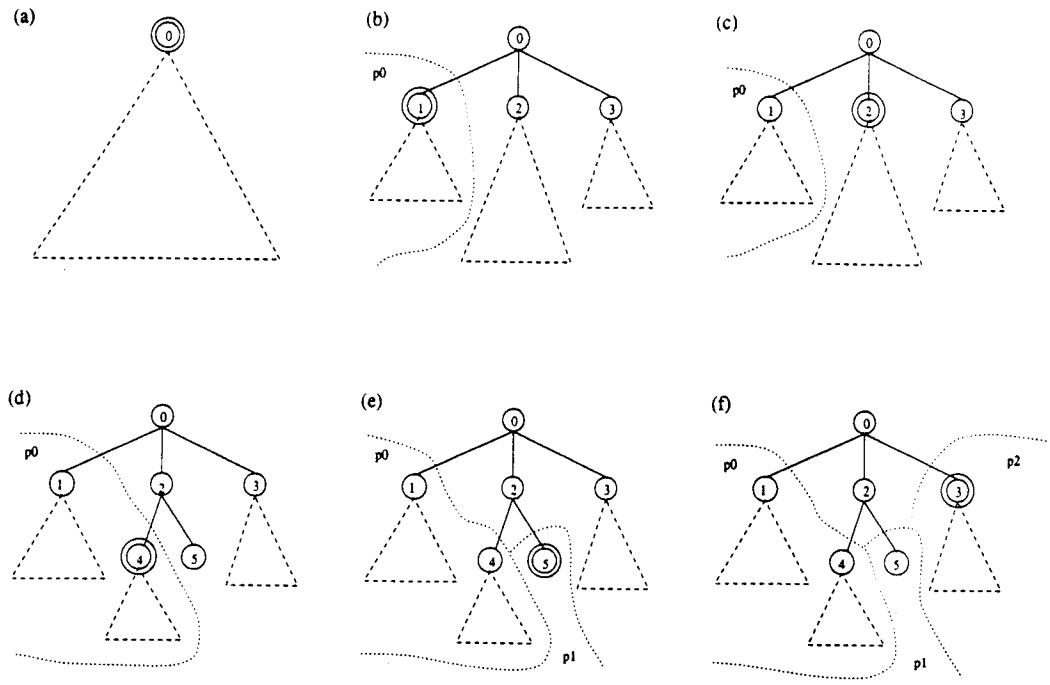


Figure 13: A tree partitioning example. (a) The partition-creation traversal starts at the root. (b) Nodes are visited and added to the current partition if their subtree fits. (c) When a subtree is too large to fit, (d) the traversal descends into the subtree. (e) Alternatively, the partition is closed and work begins on a new partition. (f) The process continues until the traversal is complete.

and the traversal continues at Node 2 (Figure 13(c)). The cost of the subtree rooted at Node 2 is too large to add to p0, so the algorithm descends to an offspring of Node 2 (Figure 13(d)). Assuming Node 4 fits in p0, the traversal continues with the next offspring of Node 2 (Figure 13(e)). Node 5 is a terminal node whose cost is larger than the available space in p0, so the decision is made to close p0 and begin a new partition, p1. As shown (Figure 13(f)), Node 5 is very expensive, and when the traversal is continued at Node 3, p1 must be closed and work continues with partition p2.

Our partitioning algorithm is similar in spirit to that of Farhat's Automatic finite element decomposer [15]. Farhat essentially performs a breadth-first search of the mesh, accumulating elements into partitions. Subdomains are accumulated during the search, and each is closed in turn when its cardinality reaches the number of elements divided by the number of processors. This is directly analagous to closing partitions in the tree algorithm. Also analagous is the possibility of subdomains (partitions) which are not single connected components. However, the similarity ends with the hierarchical nature of the tree traversal. Larger scale information is available to the tree algorithm. With

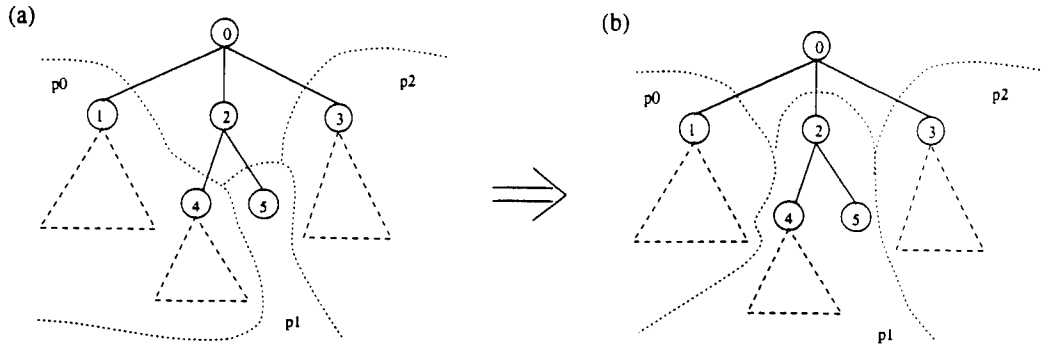


Figure 14: Iterative rebalancing of tree-based partitions. The subtree rooted at Node 4 (a) has been shifted from p0 to p1 (b) to relieve a load imbalance. The new root of p1 is Node 2, the common parent of Nodes 4 and 5.

this information, at each step it may add larger and more compact pieces of mesh to a partition, reducing the likelihood of thin partitions with large surface area.

The tree-traversal partitioning algorithm may easily be extended for use with a parallel adaptive environment. An initial partitioning is made using the serial algorithm described above. As the numerical solution advances in time, h- and/or p-refinement introduces a load imbalance. To obtain a new partitioning, let each processor compute its subtree costs using the serial traversal algorithm within its domain. This step requires no interprocessor communication. An inexpensive parallel prefix operation may be performed on the processor-subtree totals to obtain a global cost structure. This information enables a processor to determine where its local tree traversal is located in the global traversal.

Now, following the serial procedure, each processor may traverse its subtrees to create partitions. A processor determines the partition number to start working on based on the total cost of processors preceeding it. Each processor starts counting with this prefix cost and traverses its subtrees adding the cost of each visited node to this value. Partitions end near cost mutiples of N/P, where N is the total cost and P is the number of processors. Exceeding a multiple of N/P during the traversal is analagous to exceeding the optimal partition size in the serial case and the same criteria may be used to determine where to end partitions. When all processors finish their traversals, each subtree (and its associated data) is assigned to a new partition and may be migrated to its new location. Migration may be done using global communication; however, on some architectures, it may be more efficient to move data via simultaneous processor shift operations. This linear communication pattern is made possible by the one-dimensional nature of the partition traversal.

While the cost of computing the new partition is small, the cost of data

movement is likely to be high and it would be desirable to amortize this by tolerating small imbalances. A strategy to delay the need for complete repartitioning would simply shift partition boundaries, thus, migrating subtrees from a processor P_n to its neighbors P_{n-1} and P_{n+1}. If, for example, processor P_n seeks to transfer cost m to P_{n-1}, it simply traverses its subtrees accumulating their costs until it reaches m. The nodes visited comprise a subtree which may be transferred to P_{n-1} and which is contiguous with the subtrees in P_{n-1}. Likewise, if P_n desires to transfer work to P_{n+1}, the reverse traversal could remove a subtree from the trailing part of P_n. Consider, as an example, the subtree rooted at Node 4 of Figure 14(a) and suppose that its cost has increased through refinement. In Figure 14(b), we show how the partition boundary may be shifted to move the subtree rooted at Node 4 to partition p1. The amount of data to be moved from processor to processor may utilize a relaxation algorithm or the tiling procedure discussed in Sections 2 and 3.

Example 4.1 Performance results obtained by applying the tree-based mesh partitioning algorithm to various three-dimensional irregular meshes are presented in Figures 15 and 16. The meshes were generated by the Finite Octree mesh generator [29]. "Airplane" is a 182K-element mesh of the volume surrounding a simple airplane [13]. "Copter" is a 242K-element mesh of the body of a helicopter [13]. "Onera," "Onera2," and "Onera3" are 16K-, 70K-, and 293K-element meshes, respectively, of the space surrounding a swept, untwisted Onera-M6 wing which has been refined to resolve a bow shock [14]. "Cone" is a 139K-element mesh of the space around a cone having a 10° half-angle and which also has been refined to resolve a shock.

The quality of a partition has been measured in Figure 15 as the percent of element faces lying on inter-partition boundaries relative to the total number of faces of the mesh. The graph displays these percentages as a function of the optimal partition size. In all cases the cost variance between the partitions is very small (about as small as the maximum cost of a leaf octant). The proportion is, in a sense, the total surface area that partitions hold in common. Smaller ratios require less communication relative to the amount of local data access. This measure is closely related to the number of "cuts" that the partition creates [24, 20, 30]; however, we have chosen to normalize by the total number of faces in order to compare partition quality over a wide range of mesh sizes and number of partitions.

The data of Figure 15 show the expected behaviour that the interface proportion approaches zero as the partition size increases (due to the number of partitions approaching unity). Conversely, as the optimal partition size approaches unity (due to number of partitions approaching the number of elements), the interface proportion goes to unity. The interface proportion is less than 12% when the partition size exceeds 1000 for these meshes. Interfaces

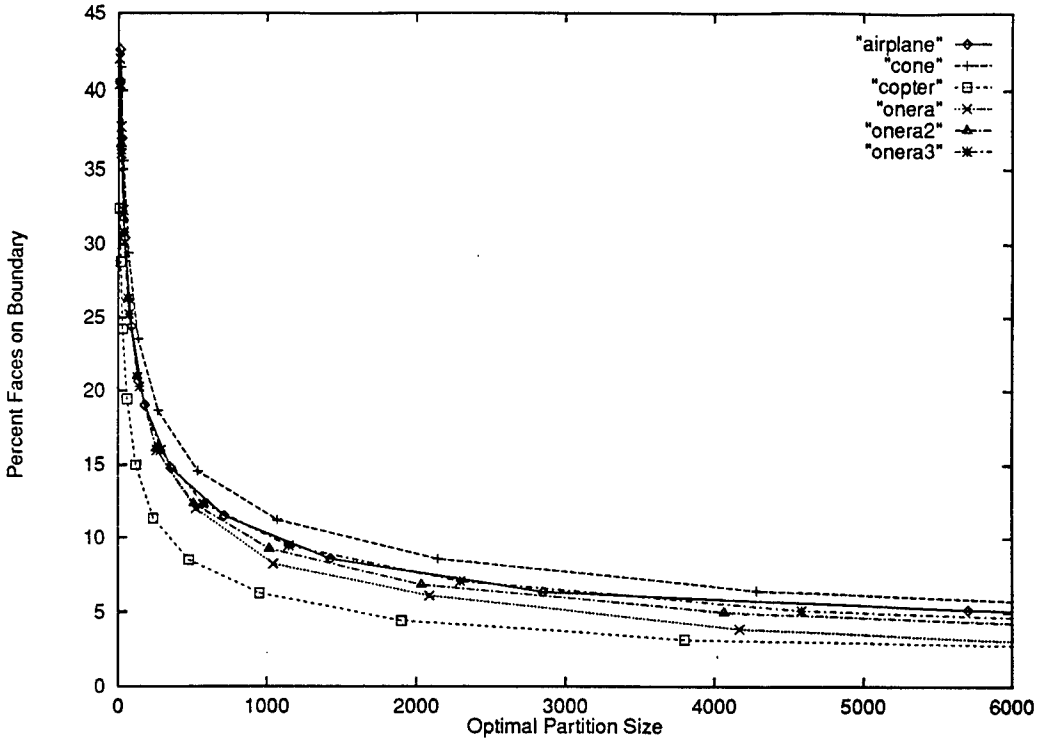


Figure 15: Global performance measure of the tree partitioning algorithm on the five meshes of Example 4.1.

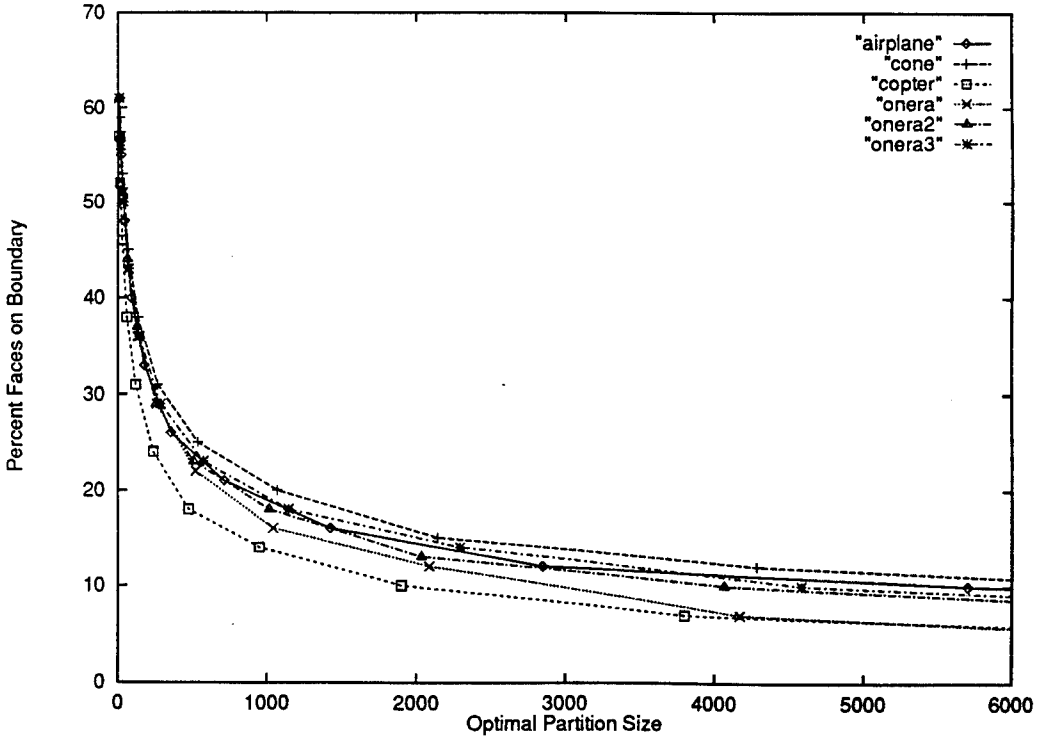


Figure 16: Local performance of the tree partitioning algorithm on the five meshes of Example 4.1.

drop to below 9% and 8%, respectively, for partition sizes of 2000 and 3000. This performance is comparable to recursive spectral bisection [23] but requires much less computation ($O(J)$ as opposed to $O(J^2)$ [26]).

The best performance occurred with the helicopter mesh, which was the only mesh of a solid object (as opposed to a flow field surrounding an object). The solid can easily be cut along its major axis to produce partitions with small inter-partition boundaries, and was included for generality. The lowest performance occurred with the cone mesh. This is most likely due to the model and shock region being conically shaped, which is somewhat at odds with the rectangular decomposition imposed by the octree.

In general, inter-partition boundaries should be less than 10%, indicating partition sizes of 2000 or more. This minimum partition size is not an excessive constraint, since a typical three-dimensional problem employing a two million-element mesh being solved on a 1024-processor computer would have about 2000 elements per processing element.

Another measure of partition quality is the percent of a partition's element faces lying on inter-partition boundaries relative to the total number of faces in that partition. This is shown in Figure 16. This number is, in a sense, the ratio of surface area to volume of a partition. For our example meshes, this measure was below 22% and 18%, respectively, for partition sizes of 1000 and 1500.

Example 4.2 In Figure 17 we show partitions of several meshes from Example 4.1. The partitions exhibit a blocked structure; however, several partitions of the airplane mesh appear to be made up of disconnected components. While this is possible, although unlikely, in this case the partitions appear to be disconnected because the display is a two-dimensional slice through the three-dimensional domain.

Example 4.3 In Figure 18 we show the pressure contours of a Mach 2 Euler flow past the "Cone" mesh of Example 4.1. The solution employs the discontinuous finite element scheme [5, 8, 9, 10] with van Leer's flux vector splitting [32] and was computed on a Thinking Machines CM-5 computer with 128 processors. Several h-refinement steps were required to yield this mesh. At each iteration, elements were marked with the desired tree level (either larger for refinement, or smaller for coarsening), and a new global mesh created to satisfy these constraints. The shock surface and pressure contours are shown above; below are examples of how the mesh may be partitioned for 16 and 32 processor machines. Each color represents membership in a different partition (and, hence, residence on a different processor).

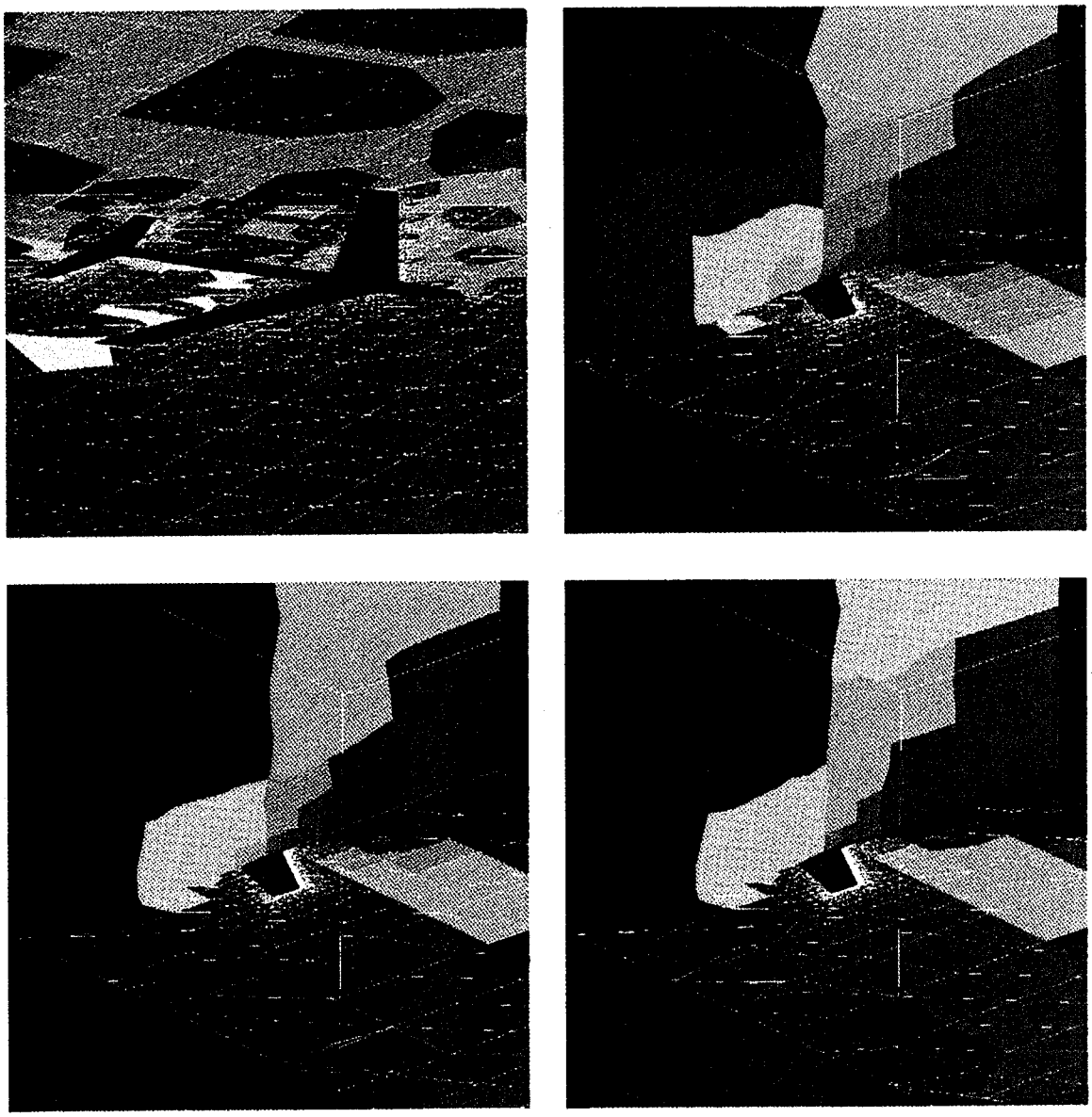


Figure 17: The airplane mesh, and three refinements of the Onera M6 wing mesh, all divided into 32 partitions. Colors denote partition membership.

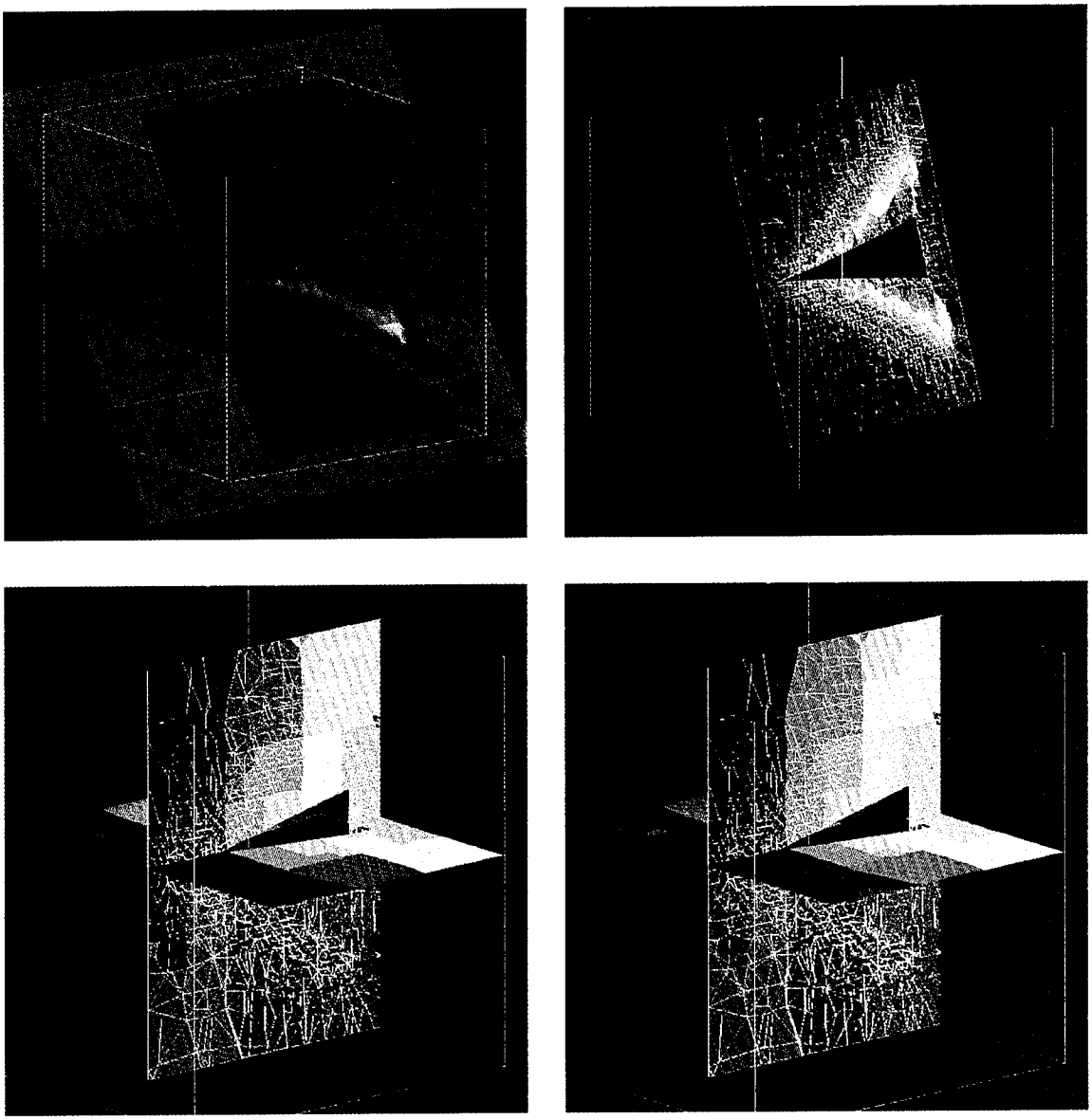


Figure 18: Shock surface and pressure contours found when computing the Mach 2 flow past a cone having a half-angle of 10° (top). Partitions of the mesh into 16 (left) and 32 (right) pieces (bottom). Colors in the top figures denote pressure levels while those of the bottom figures denote partition membership.

5 Discussion

We have described partitioning strategies that are appropriate for load balancing parallel distributed-memory computation with adaptive h- and p-refinement techniques for partial differential equations. Tiling performs local balancing within overlapping neighborhoods and we demonstrate its effective performance by using it with a local finite element technique [1, 8, 9, 10, 11] to solve two-dimensional systems of conservation laws by adaptive h- and p-refinement. The next step involves combining the h- and p-refinement procedures to develop an hp-refinement algorithm. When used with hyperbolic systems, it would appear to be appropriate to use h-refinement at points of discontinuity and p-refinement in regions of smooth flow. Appropriate combinations of h- and p-refinement at discontinuities, such as the 1:15 ratio used with elliptic problems [18], may, however, provide superior performance.

Relative to tiling, redistribution through pairwise exchanges offers the possibility of rebalancing loading more quickly and of providing a better control on the shape of partitions to reduce the communications volume. Possibilities for partition shape control involve use of orthogonal recursive bisection [6] in, say, directions of principal axes of inertia of partitions. The pairwise-exchange redistirbution procedure is being implemented for execution on a MIMD computer and development and testing will continue using three-dimensional problems in biomechanics and compressible flow as examples.

Octree-based partitioning has promise as an effective and efficient partitioning strategy that may either be used in conjunction with octree mesh generation [29] or on its own. It appears to provide a suitable means of controling communications volume based solely on the geometric decomposition of space. This aspect of the procedure must be explored more completely. Parallel partitioning techniques and incremental migration strategies for use with adaptivity are being developed. It should also be possible to combine octree partitioning with other strategies to provide additional control of communications volume. For example, octree decomposition could be used to provide an initial partition that could be continued by recursive spectral bisection [26]. Recursive spectral bisection at terminal tree node may be parallelized [24], it costs less than a global application because of the smaller partition domains and its nonlinear complexity, and it is more effective on smaller regions [23].

Theoretical issues associated with each algorithm must be investigated. Convergence under iteration of either the tiling or pairwise exchange migration strategies must be established, as must the avoidance of limit cycles.

Comparisons between methods and with other techniques must be performed; however, the three techniques described herein are under development and are not finished products. Software developed with portability in mind is, nevertheless, being executed on diverse distributed-memory platforms.

With an aim of unifying our research effort and of performing explicit comparisons, we find ourselves heading for a message passing environment using the Chameleon protocol [17].

6 Acknowledgements

This research was supported by the U.S. Army Research Office Contract Number DAAL03-91-G-0215 and DAALO3-89-C-0038 with the University of Minnesota Army High Performance Computing Research Center (AHPCRC) and the DoD Shared Resource Center at the AHPCRC; by the Massively Parallel Computation Research Laboratory, Sandia National Laboratories, operated for the U.S. Department of Energy under contract #DE-AC04-76DP00789, Research Agreement AD-9585; a DARPA Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland; and the Grumman Corporate Research Center, Grumman Corporation, Bethpage, NY 11714-3580.

We also wish to thank Thinking Machines Corporation, and in particular Zdeněk Johan and Kapil Mathur, for their assistance with the CM-5.

References

[1] S. Adjerid, M. Aiffa, and J. E. Flaherty, Adaptive Finite Element Methods for Singularly Perturbed Elliptic and Parabolic Systems, in preparation.

[2] S. Adjerid, J. Flaherty, P. Moore, and Y. Wang, High-Order Adaptive Methods for Parabolic Systems, *Physica-D*, Vol. 60, 1992, pp. 94–111.

[3] D. C. Arney and J. E. Flaherty, An Adaptive Mesh Moving and Local Refinement Method for Time-Dependent Partial Differential Equations, *ACM Trans. Math. Software*, Vol. 16, 1990, pp. 48–71.

[4] I. Babuška, The p- and hp-Versions of the Finite Element Method. The State of the Art, in *Finite Elements: Theory and Appliations*, Springer-Verlag, New York, 1988.

[5] R. Biswas, K. D. Devine, and J. E. Flaherty, Parallel, Adaptive Finite Element Methods for Conservation Laws, *Appl. Numer. Math.*, to appear.

[6] M. J. Berger and S. H. Bokhari, A Partitioning Strategy for Nonuniform Problems on Multiprocessors, *IEEE Trans. Comput.*, Vol. C-36, No. 5, May, 1987, pp. 570-580.

[7] M. J. Berger and J. Oliger, Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations, *J. Comput. Phys.*, Vol. 53, 1984, pp. 484–512.

[8] B. Cockburn, S.-Y. Lin, and C.-W. Shu, TVB Runge-Kutta Local Projection Discontinuous Galerkin Finite Element Method for Conservation Laws III: One-Dimensional Systems, *J. Comput. Phys.*, Vol. 84, 1989, pp. 90–113.

[9] B. Cockburn and C.-W. Shu, TVB Runge-Kutta Local Projection Discontinuous Galerkin Finite Element Method for Conservation Laws II: General Framework, *Math. Comp.*, Vol. 52, 1989, pp. 411–435.

[10] B. Cockburn, S.-Y. Lin, and C.-W. Shu, TVB Runge-Kutta Local Projection Discontinuous Galerkin Finite Element Method for Conservation Laws IV: The Multidimensional Case, *Math. Comp.*, Vol. 54, 1990, pp. 545–581.

[11] K. Devine, J. Flaherty, R. Loy, and S. Wheat, Parallel Partitioning Strategies for the Adaptive Solution of Conservation Laws, RPI Dept. of Comp. Sci. Tech. Rep. 94-1, 1994.

[12] P. Devloo, J. T. Oden, and P. Pattani, An h-p Adaptive Finite Element Method for the Numerical Simulation of Compressible Flow, *Comput. Methods Appl. Mech. Engrg.*, Vol. 70, 1988, pp. 203–235.

[13] S. Dey, personal communication, 1993.

[14] M. Dinar, personal communication, 1993.

[15] C. Farhat, A Simple and Efficient Automatic FEM Domain Decomposer, *Comp. and Struct.*, Vol. 28, No. 5, 1988, pp. 579–602.

[16] M. Fiedler, Algebraic Connectivity of Graphs, *Czechoslovak Math. J.*, Vol. 23, 1973, pp. 298-305.

[17] W. Gropp and B. Smith, Users Manual for the Chameleon Parallel Programming Tools, Argonne National Laboratories Tech. Rep. ANL-93/23, Argonne, 1993.

[18] W. Gui and I. Babuška, The h-, p- and hp- Versions of the Finite Element Method in One Dimension. Part I: The Error Analysis of the p Version. Part II: The Error Analysis of the h and hp Versions. Part III: The Adaptive hp Version, to appear in *Numerische Mathematik.*

[19] S. W. Hammond, Mapping Unstructured Grid Computations to Massively Parallel Computers, Ph.D. Dissertation, Computer Science Dept., Rensselaer Polytechnic Institute, Troy, 1991.

[20] B. Hendrickson and R. Leland, An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations, Sandia National Laboratories Tech. Rep. SAND92-1460, Albuquerque, 1992.

[21] B. Hendrickson and R. Leland, Multidimensional Spectral Load Balancing, Sandia National Laboratories Tech. Rep. SAND93-0074.

[22] J. Jaja, *An Introduction to Parallel Algorithms*, Addison-Wesley Publishing Company, Reading, 1992.

[23] Z. Johan, personal communication, 1993.

[24] Z. Johan, K. Mathur, and S. L. Johnsson, An Efficient Communication Strategy for Finite Element Methods on the Connection Machine CM-5 System, Thinking Machines Tech. Rep. No. 256, May, 1993. *Submitted to Computer Methods in Applied Mechanics and Engineering.*

[25] E. Leiss and H. Reddy, Distributed Load Balancing: Design and Performance Analysis, *W. M. Keck Research Computation Laboratory*, Vol. 5, 1989, pp. 205-270.

[26] A. Pothen, H. Simon, and K.-P. Liou, Partitioning Sparse Matrices with Eigenvectors of Graphs, *SIAM Journal of Matrix Analysis and Applications*, Vol. 11, 1990, pp. 430-452.

[27] E. Rank and I. Babuška, An Expert System for the Optimal Mesh Design in the *hp*-Version of the Finite Element Method, *Intl. Jrnl. Num. Meth. in Engng.*, Vol. 24, 1987, pp. 2087–2106.

[28] H. N. Reddy, On Load Balancing, Ph.D. Dissertation, Dept. Comp. Sci., Univ. of Houston, Houston, TX, 1989.

[29] M. S. Shephard and M. K. Georges, Automatic Three-Dimensional Mesh Generation by the Finite Octree Technique, *Int. J. Numer. Meths. Engng.*, Vol. 32, No. 4, 1991, pp. 709-749.

[30] H. D. Simon, Partitioning of Unstructured Problems for Parallel Processing, *Comput. Systs. Engng.*, Vol. 2, 1991, pp. 135–148.

[31] B. K. Szymanski and A. Minczuk, A Representation of a Distribution Power Network Graph, *Archiwum Elektrotechniki*, Vol. 27, No. 2, 1978, pp. 367-380.

[32] B. Van Leer, Flux Vector Splitting for the Euler Equations, ICASE Report. No. 82-30, *Inst. Comp. Applics. Sci. Engng.*, NASA Langley Research Center, Hampton, 1982.

[33] C. Walshaw and M. Berzins, Dynamic Load-Balancing for PDE Solvers on Adaptive Unstructured Meshes, Preprint, School of Computer Studies Tech. Rep., University of Leeds, 1992.

[34] S. R. Wheat, A Fine Grained Data Migration Approach to Application Load Balancing on MP MIMD Machines, Ph.D. Dissertation, Dept. Comp. Sci., Univ. of New Mexico, Albuquerque, NM, 1992.

[35] S. R. Wheat, K. D. Devine, and A. B. Maccabe, Experience with Automatic, Dynamic Load Balancing and Adaptive Finite Element Computation, *Proc. of Hawaii International Conference on System Sciences*, January, 1994, to appear.

[36] V. G. Vizing, On an estimate of a chromatic class of a multigraph, *Proc. Third Siberian Conf. on Mathematics and Mechanics*, Tomsk, 1964.